A bug collision tale

The inside story of our CVE-2019-2025 exploit

Blue Frost Security

Eloi Sanfelix
@esanfelix

Binder – Android's Inter Process Comm.

- Core IPC mechanism in Android
  - The kernel driver provides the transport mechanism
  - Context manager (service_manager) ~= service directory
  - Libraries provide transparent RPC and object serialization
- Accessible from every process
  - Potential for sandbox escape!

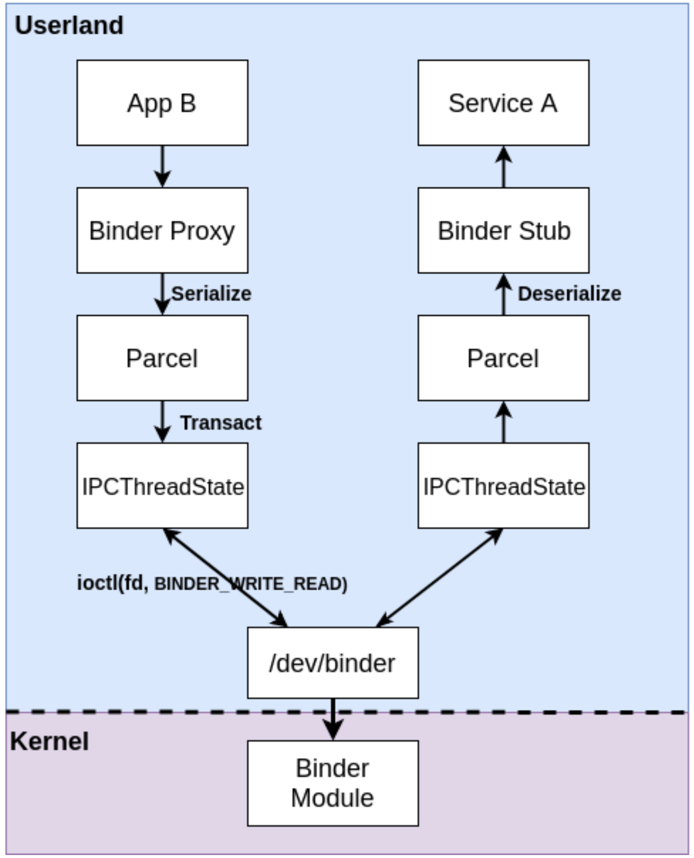
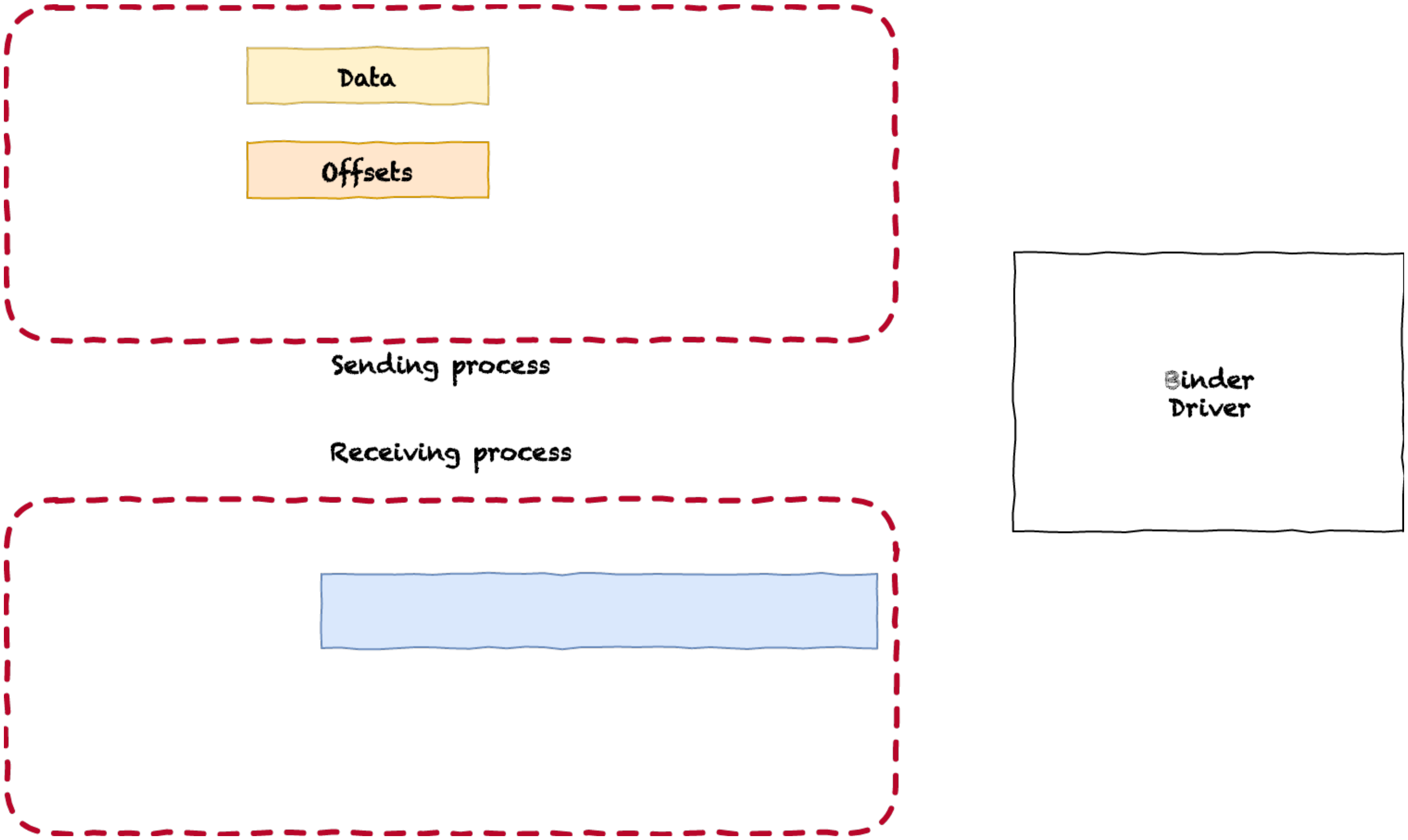
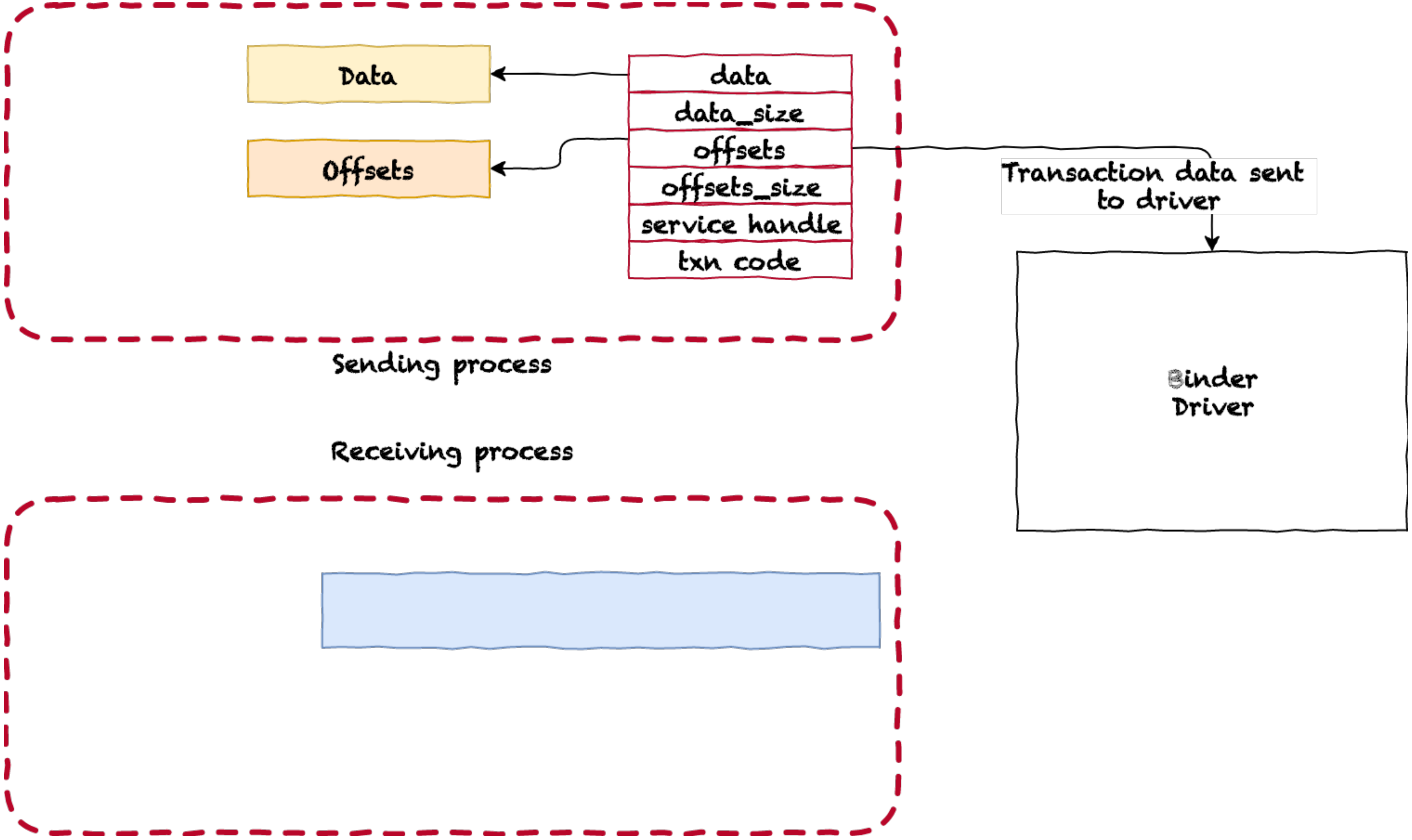


Diagram source: https://www.synacktiv.com/posts/systems/binder-transactions-in-the-bowels-of-the-linux-kernel.html

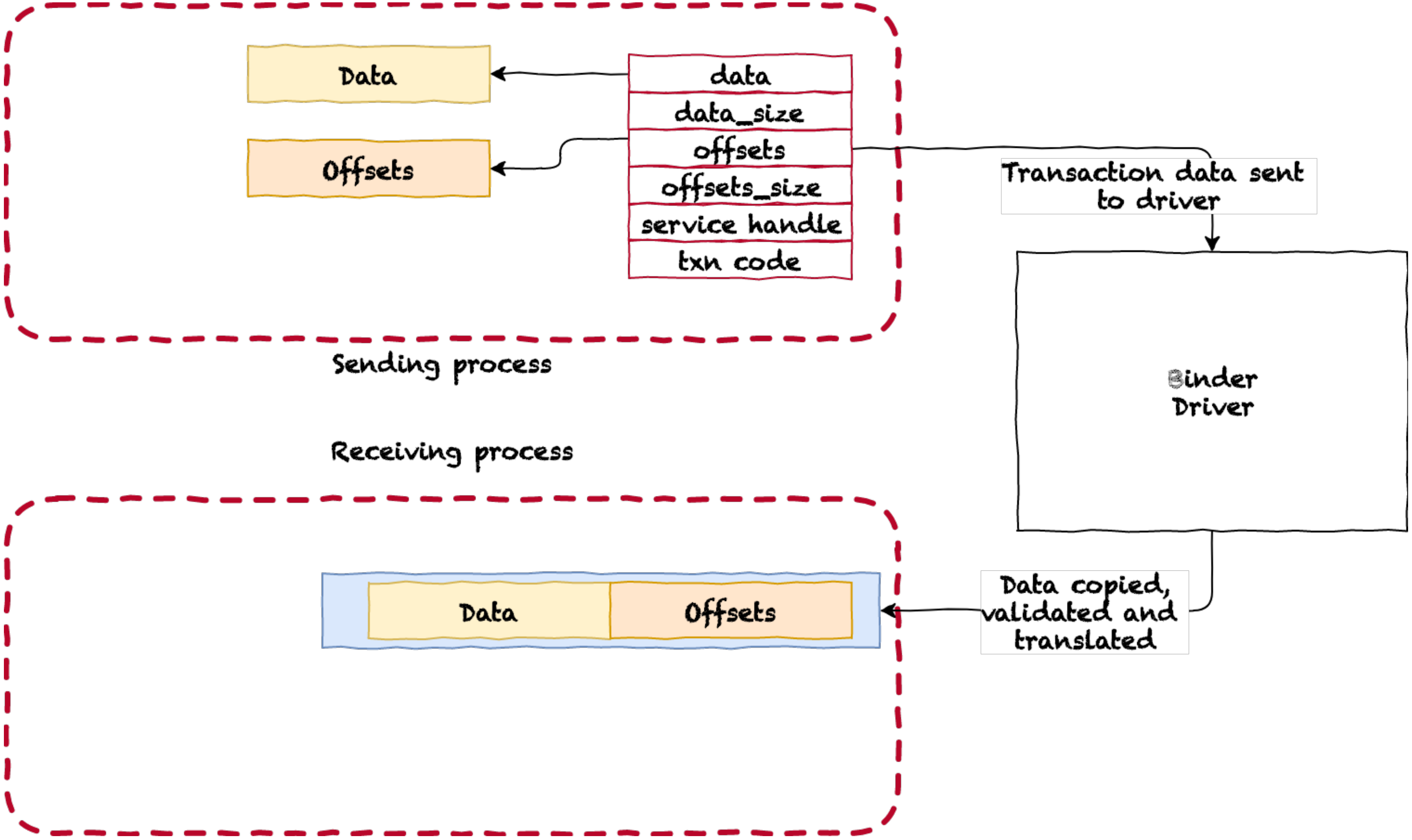
Binder transactions



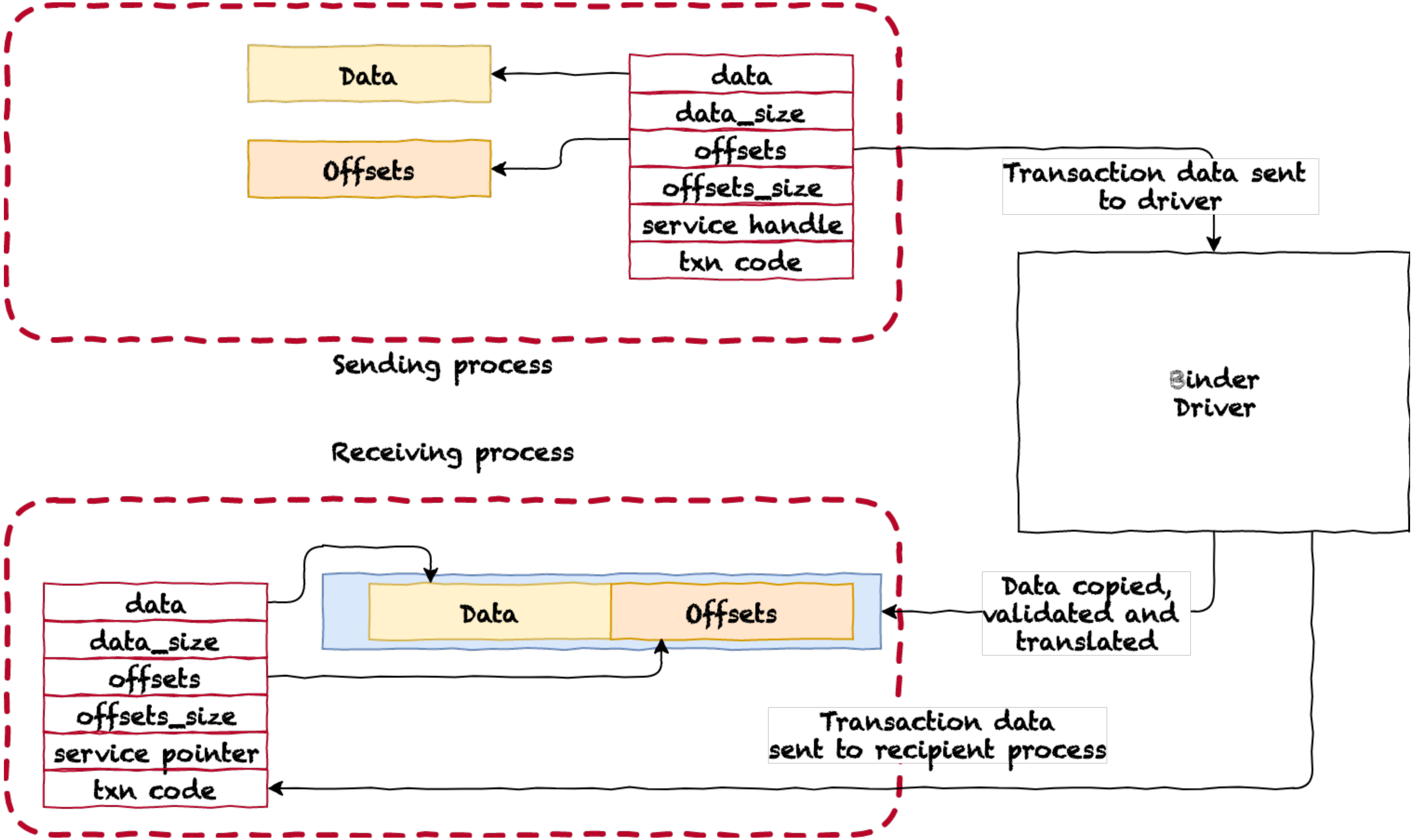
Binder transactions



Binder transactions



Binder transactions



What is in a transaction?

- Data and opaque buffers

- Binder objects
  - Represented by *struct binder_node,* reference counted
  - Translated into opaque handles when crossing process boundaries
  - Received transactions belong to a binder object (*target_node*)

- Binder handles
  - Unique at process level, relate to a remote *binder_node*
  - Translated back into objects if owned by the receiving process
  - Recipient of a transaction also identified by a handle

- File descriptors and File descriptor arrays
  - Moved from sender to receiver

Binder buffer cleanup

- Cleanup performed when recipient done with the buffer
  - Release reference to target node
  - Release references to binder objects/handles in the buffer
  - Close any file descriptor array in the buffer

- Allows for the buffer space to be reused for new transactions

The bugs – Crash log #1

```
BUG: KASAN: use-after-free in binder_thread_write+0x3dab/0x3e90 drivers/android/binder.c:3485
Write of size 8 at addr ffff88006c6adad0 by task

CPU: 0 PID: 13188 Comm:               Not tainted 4.19.0-rc1+ #1
Hardware name: QEMU Standard PC (i440FX + PIIX, 1996), BIOS 1.10.2-1ubuntu1 04/01/2014
Call Trace:
 __dump_stack lib/dump_stack.c:77 [inline]
 dump_stack+0xf0/0x1ae lib/dump_stack.c:113
 print_address_description.cold.3+0x9/0x225 mm/kasan/report.c:256
 kasan_report_error mm/kasan/report.c:354 [inline]
 kasan_report.cold.4+0x66/0x9a mm/kasan/report.c:412
 binder_thread_write+0x3dab/0x3e90 drivers/android/binder.c:3485
 binder_ioctl_write_read+0x3eb/0xa80 drivers/android/binder.c:4459
 binder_ioctl+0x86b/0x1269 drivers/android/binder.c:4599

Allocated by task 1454:
 set_track mm/kasan/kasan.c:460 [inline]
 kasan_kmalloc+0xbf/0xe0 mm/kasan/kasan.c:553
 kmem_cache_alloc_trace+0xe2/0x1a0 mm/slub.c:2734
 kmalloc include/linux/slab.h:513 [inline]
 kzalloc include/linux/slab.h:707 [inline]
 binder_transaction+0x14ab/0x8b20 drivers/android/binder.c:2907
```

Root cause: Race condition

Thread 1

```c
t->buffer->allow_user_free = 1;
if (...){
    ...
} else {
    binder_free_transaction(t);
}


static void binder_free_transaction(...)
{
    if (t->buffer)
        t->buffer->transaction = NULL;
    kfree(t);
}
```

Thread 2

```c
if (!buffer->allow_user_free) {
    binder_user_error(...);
    break;
}
    binder_debug(...);


if (buffer->transaction) {



    buffer->transaction->buffer = NULL;
    buffer->transaction = NULL;
}
```

Problems

1. Very narrow race condition (just a few instructions)
2. No time for reclaiming freed buffer
3. Can only write NULL at a certain offset

→Decided against pursuing this one

The bugs – Crash log #2

```
kernel BUG at drivers/android/binder_alloc.c:601!
invalid opcode: 0000 [#1] SMP KASAN PTI
CPU: 1 PID: 27808 Comm:                 Not tainted 4.19.0-rc1+ #1
Hardware name: QEMU Standard PC (i440FX + PIIX, 1996), BIOS 1.10.2-1ubuntu1 04/01/2014
RIP: 0010:binder_free_buf_locked+0x5f2/0x7d0 drivers/android/binder_alloc.c:601
Code: 64 ad 43 fe 0f 0b 48 c7 c7 00 34 0c 85 e8 dc 36 00 ff e8 51 ad 43 fe 0f 0b 48 c7 c7
binder: BINDER_SET_CONTEXT_MGR already set
RSP: 0018:ffff88004f21f378 EFLAGS: 00010216
RAX: 0000000000040000 RBX: ffff88006863db80 RCX: ffffc90006b50000
RDX: 0000000000000785 RSI: ffffffff830099f2 RDI: ffff88006863dbb0
RBP: ffff88006bda4620 R08: ffff8800689ba640 R09: ffffed0009e43e6e
R10: ffffed0009e43e6d R11: ffff88004f21f36f R12: 0000000000000028
R13: 0000000000000028 R14: 0000000000000000 R15: ffff88006bda4660
FS:  00007fd7f3264700(0000) GS:ffff88006d100000(0000) knlGS:0000000000000000
CS:  0010 DS: 0000 ES: 0000 CR0: 0000000080050033
binder: 27794:27850 ioctl 40046207 20000000 returned -16
CR2: 00007fd7f3242db8 CR3: 000000006a46c000 CR4: 00000000000006e0
Call Trace:
 binder_alloc_free_buf+0x20/0x30 drivers/android/binder_alloc.c:650
 binder_thread_write+0xa46/0x3e90 drivers/android/binder.c:3509
 binder_ioctl_write_read+0x3eb/0xa80 drivers/android/binder.c:4459
 binder_ioctl+0x86b/0x1269 drivers/android/binder.c:4599
 vfs_ioctl fs/ioctl.c:46 [inline]
 file_ioctl fs/ioctl.c:501 [inline]
```

Root cause: Race condition #2

Thread 1

```
t->buffer = binder_alloc_new_buf(...);


if (IS_ERR(t->buffer)) {
    ...
}




t->buffer->allow_user_free = 0;

...
```

Thread 2

```


buffer =
binder_alloc_prepare_to_free(...);


if (!buffer->allow_user_free) {
    binder_user_error(...);
    break;
}
    binder_debug(...);
    ...
```

allow_user_free cleared without any locks → can free transaction buffer prematurely


ARE
YOU
READY?

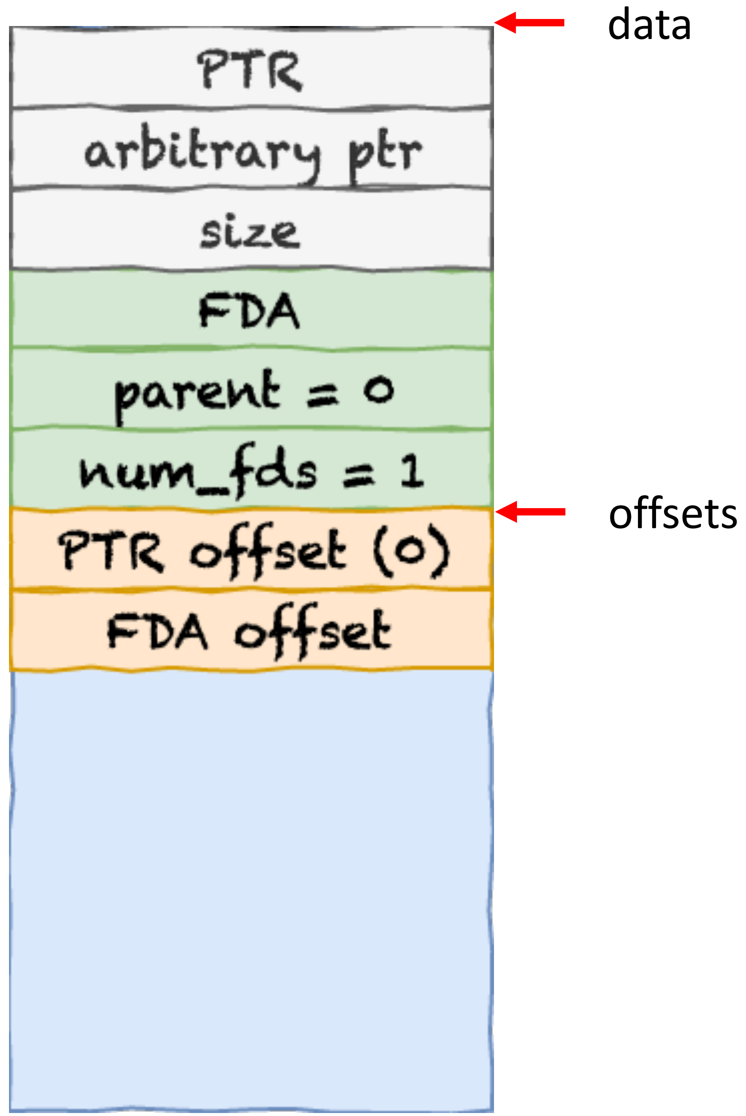

CIUDAD CULTURAL
e.ko
ekoparty 2018
26 27 28 DE SEPTIEMBRE

Promising idea

1. Send fake objects as data

2. Receive transaction and free buffer
   - Freed buffer contains arbitrary content now

3. Create new transaction and trigger bug
   - Force sending path to fail early
   - Free path will process fake objects!



What does this do?

```
static void binder_transaction_buffer_release(...)
{
  /* ... */
  for (offp = off_start; offp < off_end; offp++) {
    /* ... */

      case BINDER_TYPE_FDA: {
        /* ... */

        fda = to_binder_fd_array_object(hdr);
        parent = binder_validate_ptr(...);
        /* ... */

        parent_buffer = parent->buffer - <some_offset>;

        /* ... */
        fd_array = (u32 *)(parent_buffer + (uintptr_t)fda->parent_offset);
        for (fd_index = 0; fd_index < fda->num_fds; fd_index++)
          task_close_fd(proc, fd_array[fd_index]);
```

parent_buffer
semi-arbitrary

Crash reading from
semi-controlled address

Meaning what?

- We can force release of fake objects
- A valid FDA allows to close arbitrary file descriptors
  - Useful to know when we triggered the race
- Binder objects will cause refcount decrements
  - Decrement to zero to cause use-after-free

The high level plan

1. Prefill transaction buffer with fake objects

2. Free transaction buffer (*allow_user_free = 1* now)

3. Initiate transaction with data and offsets
   - Reuses previous buffer with fake objects
   - Setup to fail quickly (e.g. unaligned offset size)

4. Race transaction to cause free of buffer with fake objects
   1. Monitor file descriptors to know when we won the race
   2. Tinker with reference counts to cause *binder_node* use-after-frees

5. Somehow get kernel read/write from there and profit

But …

```
t->buffer->debug_id = t->debug_id;
t->buffer->transaction = t;
t->buffer->target_node = target_node;
trace_binder_transaction_alloc_buf(t->buffer);
off_start = (binder_size_t *)(t->buffer->data +
      ALIGN(tr->data_size, sizeof(void *)));
offp = off_start;


if (copy_from_user(t->buffer->data, (const void __user *)(uintptr_t)
  tr->data.ptr.buffer, tr->data_size)) {
    ...
}
```

Free before this point and buffer will be NULL → CRASH!

... and ...

```c
static void binder_free_buf_locked(...)
{
    size_t size, buffer_size;
    buffer_size = binder_alloc_buffer_size(alloc, buffer);


    ...

    BUG_ON(buffer->free);
    BUG_ON(size > buffer_size);
    BUG_ON(buffer->transaction != NULL);
    BUG_ON(buffer->data < alloc->buffer);
    BUG_ON(buffer->data > alloc->buffer + alloc->buffer_size);
```

These crash the kernel once *binder_free_buf_locked()* is called a second time ☹

… and …

- We need to be on both sides of the transaction
  - For synchronization
  - For knowing the identifiers of binder nodes (pointers)

- This is not something normally allowed on Android
  - You need handles to talk to a process
  - Binder will never give you handles to your own process
  - Still possible with some tricks (e.g. *ITokenManager* in *hwbinder*)
  - Though this limits us to non-sandboxed processes

… and on top of that

- We need to deal with quite some mitigations
  - Kernel ASLR
  - PAN and PXN (AArch64 equivalents to SMEP/SMAP)
  - Control Flow Integrity (clang’s on Pixel 3, Samsung’s on Galaxy S9)
  - Runtime Kernel Protection on Galaxy S9


Land of sun and
root shells here

The roadmap

- ☐ Avoid the NULL pointer exception
- ☐ Avoid the BUG_ON crashes
- ☐ Use use-after-free to bypass KASLR
  - Accounting for PAN, PXN, CFI, etc.
- ☐ Use use-after-free to get read/write
- ☐ Cleanup and get root

The roadmap

- ☐ Avoid the NULL pointer exception
- ☐ Avoid the BUG_ON crashes
- ☐ Use use-after-free to bypass KASLR
  - Accounting for PAN, PXN, CFI, etc.
- ☐ Use use-after-free to get read/write
- ☐ Cleanup and get root

Avoiding the NPE

Thread 1

```
t->buffer = binder_alloc_new_buf(...);


if (IS_ERR(t->buffer)) {
    ...
}





t->buffer->allow_user_free = 0;

...
```

Thread 2

```


buffer =
binder_alloc_prepare_to_free(...);


if (!buffer->allow_user_free) {
    binder_user_error(...);
    break;
}
    binder_debug(...);
    ...
```

Run on a faster CPU

Run on a slower CPU

Avoiding the NPE (II)

- This kind of works but requires tweaking per device
    - E.g. I need different settings for Pixel 1, Pixel 3 and S9
- My colleague used mutex contention instead
    - Binder allocator uses mutex to synchronize threads
    - Unlocking a mutex with waiters can cause scheduler calls
    - This helps evict the thread off the CPU and gives us some time

The roadmap

- ☑ Avoid the NULL pointer exception
- ☐ Avoid the BUG_ON crashes
- ☐ Use use-after-free to bypass KASLR
  - Accounting for PAN, PXN, CFI, etc.
- ☐ Use use-after-free to get read/write
- ☐ Cleanup and get root

The roadmap

☑ Avoid the NULL pointer exception

☐ Avoid the BUG_ON crashes

☐ Use use-after-free to bypass KASLR

- Accounting for PAN, PXN, CFI, etc.

☐ Use use-after-free to get read/write

☐ Cleanup and get root

Avoiding the BUG_ON – The ugly way

1. Make a monitor thread to check for file descriptor closing
    1. This indicates we won the race
2. Pin such thread to same CPU as receiver thread
3. Receiver: decrement reference count, then close fd
4. Monitor: when fd is closed, free object and reclaim, then cause a deadlock

→ Complete CPU hangs. Extremely ugly, but we have 8 of them ;-)

Avoiding the BUG_ON – The elegant way

- Colleague found a way to fix things from userland
  - No deadlock → cleaner solution, more stable (2 UAF instead of 3)
- Send and receive a one-way transaction in monitor thread
  - Buffer gets reused
  - Transaction is freed upon receipt, but buffer is not
  - Our thread sees an allocated buffer as expected → no BUG!

The roadmap

- ☑ Avoid the NULL pointer exception
- ☑ Avoid the BUG_ON crashes
- ☐ Use use-after-free to bypass KASLR
  - Accounting for PAN, PXN, CFI, etc.
- ☐ Use use-after-free to get read/write
- ☐ Cleanup and get root

The roadmap

☑ Avoid the NULL pointer exception

☑ Avoid the BUG_ON crashes

☐ Use use-after-free to bypass KASLR

- Accounting for PAN, PXN, CFI, etc.

☐ Use use-after-free to get read/write

☐ Cleanup and get root

Reclaiming freed objects – heap spray

- Most known generic and convenient techniques won't work
  - CONFIG_KEYS=n → no *keyctl* spraying
  - CONFIG_SYSVIPC=n → no *msgsnd* spray
  - CONFIG_USERFAULTFD=n → no easy blocking of kernel paths
- Another complication: minimum *kmalloc* object is 128 bytes
  - Everything smaller than that gets mixed with our target objects
- We settled for *sendmsg-based* spraying (known method)
  - Full control of size (with a minimum), contents and lifetime ☺
  - But requires one thread per allocated object ☹

Sendmsg spray

```
static int ___sys_sendmsg(/* ... */) {
   unsigned char ctl[sizeof(struct cmsghdr) + 20];
   /* ... */
   if (ctl_len > sizeof(ctl)) {
      ctl_buf = sock_kmalloc(sock->sk, ctl_len, GFP_KERNEL);
      if (ctl_buf == NULL)
         goto out_freeiov;
   }
   err = -EFAULT;

   if (copy_from_user(ctl_buf,
      (void __user __force *)msg_sys->msg_control,
        ctl_len))
      goto out_freectl;
    msg_sys->msg_control = ctl_buf;
/* ... */
out_freectl:
if (ctl_buf != ctl)
   sock_kfree_s(sock->sk, ctl_buf, ctl_len);
```

Controlled size to *kmalloc*

Contents copied from userland

Must block to prevent freeing object here

Leaking data

```
if (t->buffer->target_node) {
  struct binder_node *target_node = t->buffer->target_node;
  struct binder_priority node_prio;

  tr.target.ptr = target_node->ptr;
  tr.cookie =  target_node->cookie;

  node_prio.sched_policy = target_node->sched_policy;
  node_prio.prio = target_node->min_priority;
  binder_transaction_priority(...);
  cmd = BR_TRANSACTION;
}
```

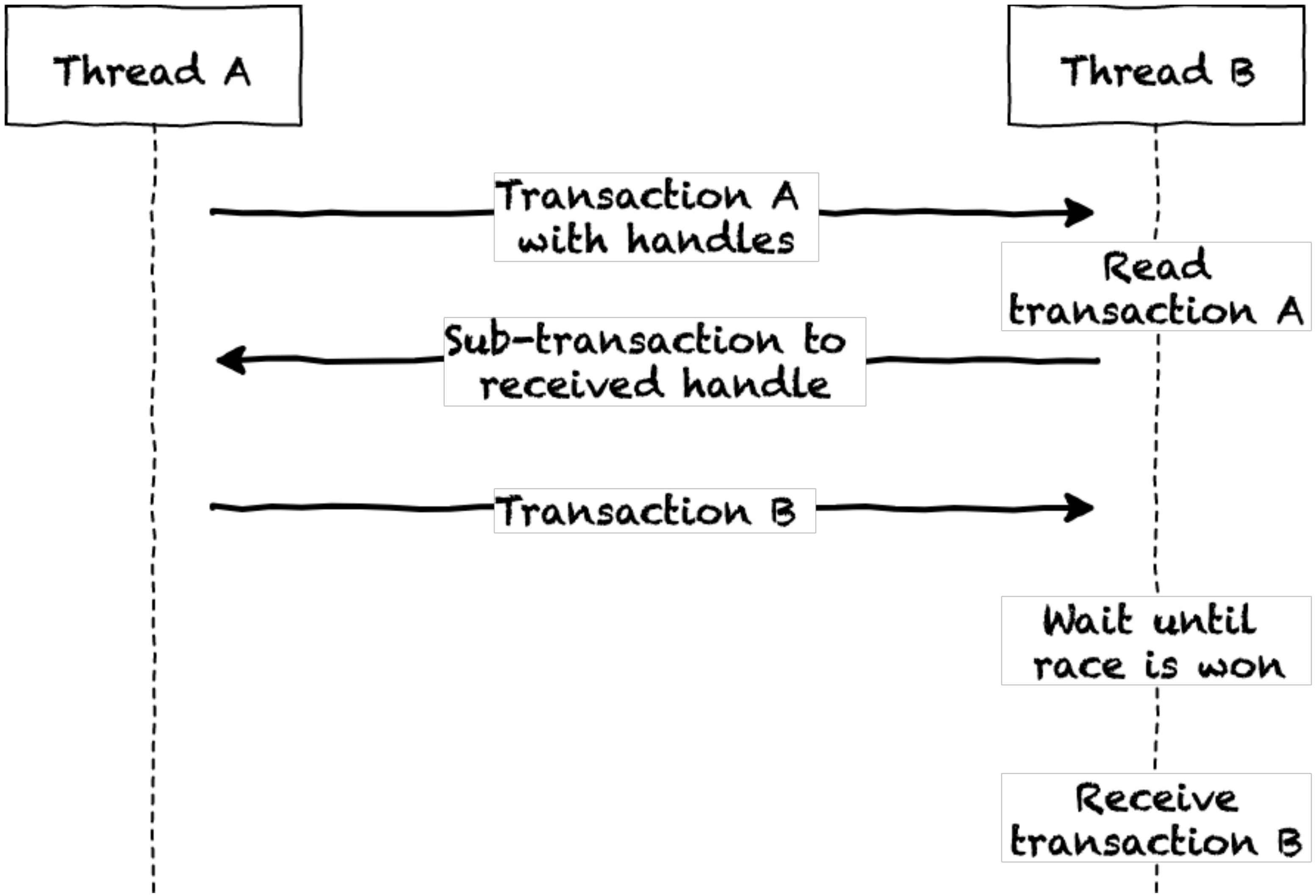
We can free this node and replace it by another object

This data gets returned to userland when the transaction is received.

Houston ...

- We need to keep the transaction pending until UAF triggered
- Transactions usually given to first looper trying to receive anything
  - But we need transactions to trigger our bug!!
- Except there can be nested transactions
  - In that case, the nested transactions go to the exact same thread
  - We thus need to make a nested transaction, wait for UAF and then receive it

Leaving pending transactions



Searching for something to leak ...

```
struct epitem {
  /* ... */

  /* List containing poll wait queues */
  struct list_head pwqlist;

  /* The "container" of this item */
  struct eventpoll *ep;

  /* List header used to link this item to the "struct file"    items list */
  struct list_head fllink;

  /* wakeup_source used when EPOLLWAKEUP is set */
  struct wakeup_source __rcu *ws;

  /* The structure that describe the interested events and the source fd */
  struct epoll_event event;
};
```

1. Can be allocated with epoll
2. Can make *as many as we want*
3. Leaks *struct file ** to userland

One step closer!

```
01-02 04:43:06.638 19465 19561 D XXX     : [*] Now waiting for ping pong to come in. func: 0x7d8a71fcfc
01-02 04:43:06.639 19465 19558 D XXX     : [*] GOT 0x4343. Playing pingpong now!
01-02 04:43:06.639 19465 19558 D XXX     : [*] Waiting for pingpong to finish
01-02 04:43:06.639 19465 19562 D XXX     : [*] ping-pong thread (19562) going ahead with transaction
01-02 04:43:06.639 19465 19562 D XXX     : [*] ping-pong thread sending transaction. using handle: 9
01-02 04:43:06.639 19465 19562 D XXX     : [*] We should have received this transaction already. Sending new one
01-02 04:43:06.640 19465 19562 D XXX     : [*] ping-pong received inner transaction.
01-02 04:43:06.640 19465 19558 D XXX     : [*] Going ahead.
01-02 04:43:06.646 19465 19557 D XXX     : [*] Reading transaction. Should be 0x2100-byte buffer @ 0x68
01-02 04:43:06.646 19465 19557 D XXX     : [*] Transaction read.
01-02 04:43:06.646 19465 19557 D XXX     : [*] extracted race_buffer: 7d5800b068
01-02 04:43:06.646 19465 19557 D XXX     : [*] Exploit thread ready to go!
01-02 04:43:06.646 19465 19558 D XXX     : [*] Buffer size: 1700 , num_fds = 1471, num offsets: 768
01-02 04:43:06.646 19465 19558 D XXX     : [*] Number of fds to close during release: 1116489. FD: 4095
01-02 04:43:07.518 19465 19465 D XXX     : [*] All threads connected.
01-02 04:43:09.521 19465 19557 D XXX     : [+] Triggering the bug
01-02 04:43:15.462 19465 19558 D XXX     : [+] racer exit!
01-02 04:43:15.468 19465 19559 D XXX     : [*] monitor fd closed
01-02 04:43:15.468 19465 19560 D XXX     : [*] Freeing 1st node
01-02 04:43:15.518 19465 19557 D XXX     : [+] exploit: race finished!
01-02 04:43:15.522 19465 19560 D XXX     : [*] We still alive in uaf(), waiting for leaked buffer
01-02 04:43:15.522 19465 19561 D XXX     : [LEAK] buffer=0x7d5800b060 code=-559038737 ptr=ffffffdf238f5ad8 cookie=ffffffdf238f5ad8
```

Solving unstable spray issues

- Leak only working ~70% of the time
- Traced *kmalloc*/*kfree* related calls
  - Using Linux kernel tracing facilities
- Reallocation thread sometimes didn’t match expectations
  - Slab may be active on a different CPU than the one we’re allocating from
- Easiest solution: allocate from all CPUs in the system
  - Now we’re leaking quite reliably!


Slow progress is better than no progress.
Anonymous
quotefancy

The roadmap

☑ Avoid the NULL pointer exception

☑ Avoid the BUG_ON crashes

☑ Use use-after-free to bypass KASLR
  - Accounting for PAN, PXN, CFI, etc.

☐ Use use-after-free to get read/write

☐ Cleanup and get root

The roadmap

☑ Avoid the NULL pointer exception

☑ Avoid the BUG_ON crashes

☑ Use use-after-free to bypass KASLR

- Accounting for PAN, PXN, CFI, etc.

☐ Use use-after-free to get read/write

☐ Cleanup and get root

Corrupting data

```
if (hlist_empty(&node->refs) && !node->local_strong_refs &&
    !node->local_weak_refs && !node->tmp_refs) {
  if (proc) {
    binder_dequeue_work_ilocked(&node->work);
    rb_erase(&node->rb_node, &proc->nodes);
    binder_debug(...);
} else {
```

```
static void binder_dequeue_work_ilocked(struct binder_work *work)
{
   list_del_init(&work->entry);
}
```

Two *aa8bmo* primitives

list_del (CONFIG_DEBUG_LIST = n)

```
*(next + 8) = prev
*(prev)     = next
```

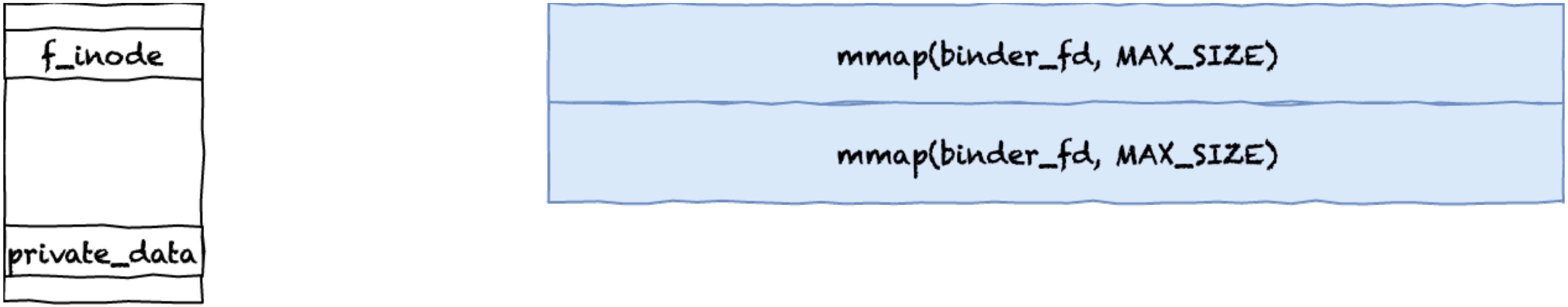
rb_erase (if rb_left is NULL)

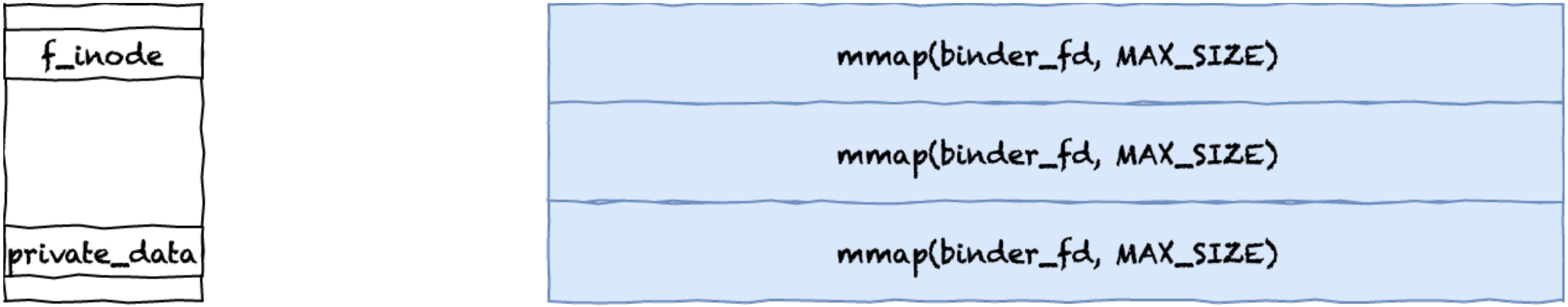
```
*(parent + 8) = rb_right
*(rb_right)   = parent
```

So what?

- Can write semi-arbitrary values to controlled address
  - Written value needs itself to be a writable address
  - There’s PAN, so it needs to be a KERNEL address too
- We only know one address so far: a *struct file **
- Solved with *binder mapping spray*


f_inode
private_data
mmap(binder_fd, MAX_SIZE)


f_inode
private_data
mmap(binder_fd, MAX_SIZE)
mmap(binder_fd, MAX_SIZE)


f_inode
private_data
mmap(binder_fd, MAX_SIZE)
mmap(binder_fd, MAX_SIZE)
mmap(binder_fd, MAX_SIZE)


f_inode
private_data
mmap(binder_fd, MAX_SIZE)
mmap(binder_fd, MAX_SIZE)
mmap(binder_fd, MAX_SIZE)
mmap(binder_fd, MAX_SIZE)
mmap(binder_fd, MAX_SIZE)


f_inode
private_data
mmap(binder_fd, MAX_SIZE)
mmap(binder_fd, MAX_SIZE)
mmap(binder_fd, MAX_SIZE)
mmap(binder_fd, MAX_SIZE)
mmap(binder_fd, MAX_SIZE)

Status check

- We can place arbitrary data at a known location (and modify it)
  - Bye bye PAN
- We can leak a *struct file ** and corrupt *private_data* and *f_inode*
- We can fake those structures in our binder mapping
  - But we need to figure out which one of the mappings is the good one
  - Easy: scan through them and find which one was modified!

Getting arbitrary read through *f_inode*

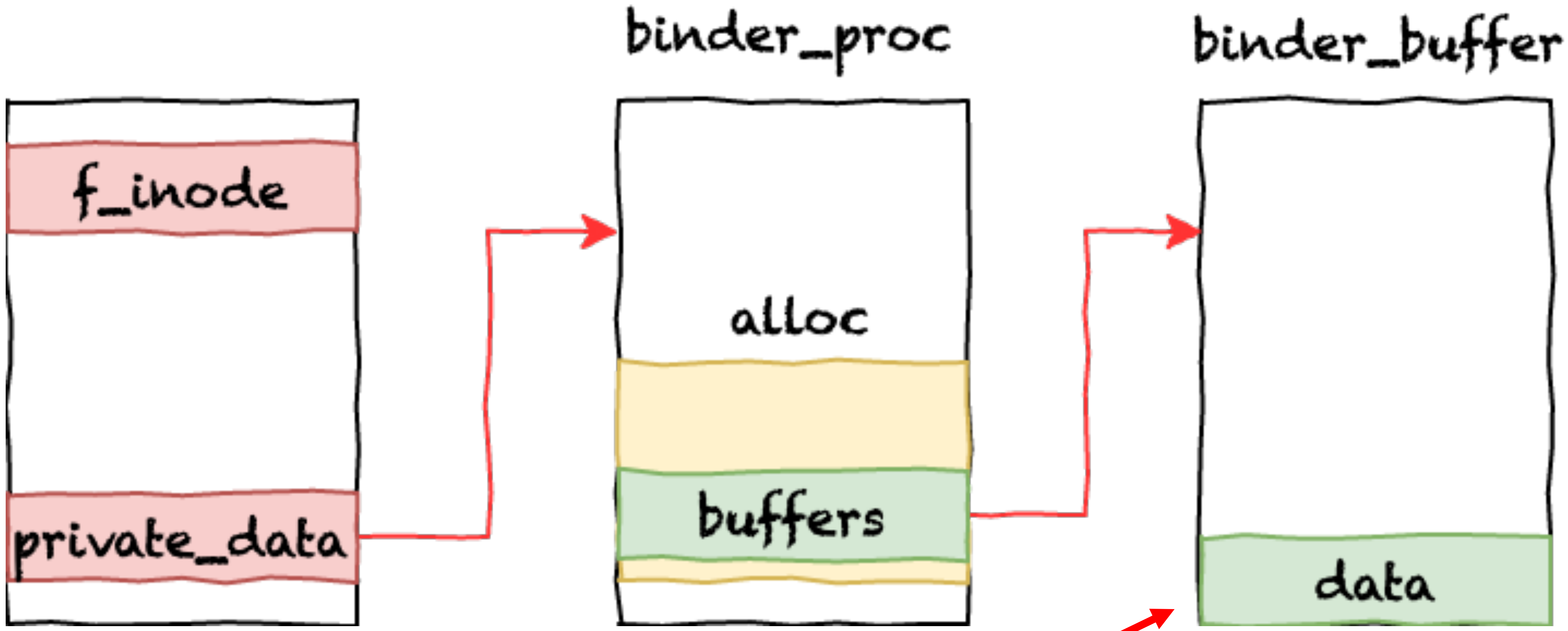
```
int do_vfs_ioctl(struct file *filp, unsigned int fd, unsigned int cmd,
    unsigned long arg)
{
  int error = 0;
  int __user *argp = (int __user *)arg;
  struct inode *inode = file_inode(filp);

  switch (cmd) {

    /* ... */

    case FIGETBSZ:
        return put_user(inode->i_sb->s_blocksize, argp);
```

1. Set *i_sb* to arbitrary address.
2. *ioctl(fd, FIGETBSZ)* to read 4 bytes

Status check

- We can leak a *struct file ** and corrupt *private_data* and *f_inode*
  - We also learn some pointers as a side effect ☺
- Can read arbitrary data
  - Bye bye KASLR
- *All* we need is an arbitrary write now!

Arbitrary writes through binder transactions



Sending a transaction will write to this address

The roadmap

☑ Avoid the NULL pointer exception

☑ Avoid the BUG_ON crashes

☑ Use use-after-free to bypass KASLR

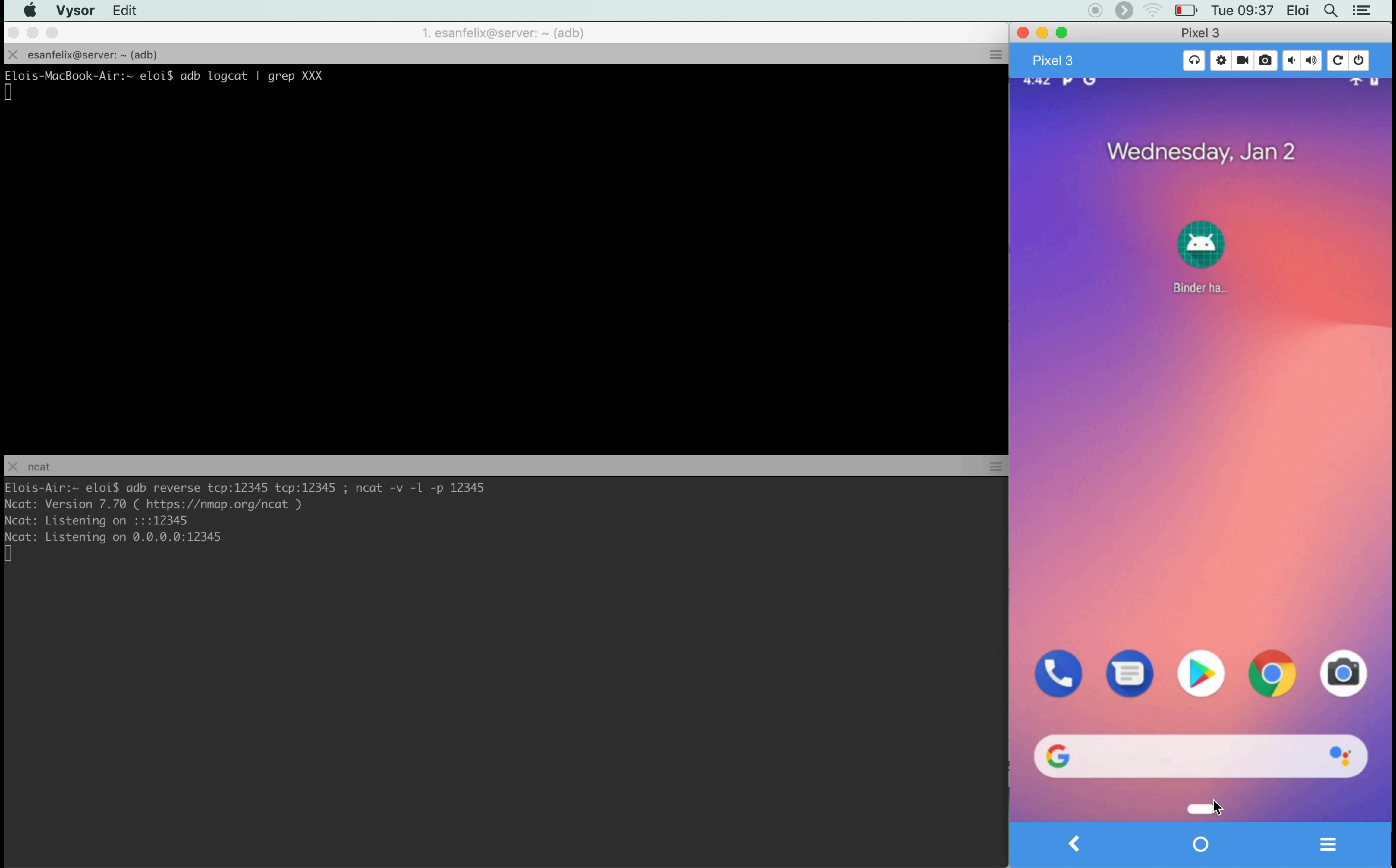
- Accounting for PAN, PXN, CFI, etc.

☑ Use use-after-free to get read/write

☐ Cleanup and get root

And now?

1. Set *selinux_enforcing* to zero
2. Find *init* and steal its credentials
3. Clean things up
   - Release deadlocked CPU!
   - Clear references to replacement objects to prevent double-free
4. Start reverse root shell


Vysor
Edit
Tue 09:37
Eloi
1. esanfelix@server: ~ (adb)
esanfelix@server: ~ (adb)
EloIs-MacBook-Air:~ eloi$ adb logcat | grep XXX
ncat
Elois-Air:~ eloi$ adb reverse tcp:12345 tcp:12345 ; ncat -v -l -p 12345
Ncat: Version 7.70 (https://nmap.org/ncat)
Ncat: Listening on :::12345
Ncat: Listening on 0.0.0.0:12345
Pixel 3
Pixel 3
Wednesday, Jan 2
Binder ha...

The roadmap

☑ Avoid the NULL pointer exception

☑ Avoid the BUG_ON crashes

☑ Use use-after-free to bypass KASLR

- Accounting for PAN, PXN, CFI, etc.

☑ Use use-after-free to get read/write

☑ Cleanup and get root

Are we done yet? What about Galaxy S9?

- list_del write cannot be used (CONFIG_DEBUG_LIST=y)
  - Only one *aa8bmo*
  - Used *private_data* only, found a way to get a second one from there

- Old binder driver with CVE-2017-13164
  - Leaks addresses of binder buffer → no need to spray
  - Can see changes to allocator structures in mapping → reduced NPE chances
  - binder-based write not feasible → overwrite *file_operations* and use “ROP"

- *selinux_enforcing* is read-only → cannot disable SELinux
  - Just loaded a new policy after stealing *init* credentials
  - RKP not fully implemented on S9 at that moment

But then …

From Todd Kjos <>
Subject [PATCH] binder: fix race that allows malicious free of live buffer share
Date Tue, 6 Nov 2018 15:55:32 -0800

```
Malicious code can attempt to free buffers using the
BC_FREE_BUFFER ioctl to binder. There are protections
against a user freeing a buffer while in use by the
kernel, however there was a window where BC_FREE_BUFFER
could be used to free a recently allocated buffer that
was not completely initialized. This resulted in a
use-after-free detected by KASAN with a malicious
test program.

This window is closed by setting the buffer's
allow_user_free attribute to 0 when the buffer
is allocated or when the user has previously
freed it instead of waiting for the caller
to set it. The problem was that when the struct
buffer was recycled, allow_user_free was stale
and set to 1 allowing a free to go through.

Signed-off-by: Todd Kjos <tkjos@google.com>
Acked-by: Arve Hjønnevåg <arve@android.com>
---
 drivers/android/binder.c       | 21 ++++++++++++---------
 drivers/android/binder_alloc.c | 16 ++++++----------
 drivers/android/binder_alloc.h |  3 +--
 3 files changed, 19 insertions(+), 21 deletions(-)

diff --git a/drivers/android/binder.c b/drivers/android/binder.c
index cb30a524d16d8..9f1000d2a40c7 100644
--- a/drivers/android/binder.c
+++ b/drivers/android/binder.c
@@ -2974,7 +2974,6 @@ static void binder_transaction(struct binder_proc *proc,
                t->buffer = NULL;
                goto err_binder_alloc_buf_failed;
        }
-       t->buffer->allow_user_free = 0;
        t->buffer->debug_id = t->debug_id;
```

A few months later ...

360 核心安全技术博客

- 主页 Home
- 归档 Archive
- 分类 Category
- 关于 About

The 'Waterdrop' in Android: A Binder Kernel Vulnerability

03月06, 2019

Author: Hongli Han(@hexb1n) of Qihoo 360 C0RE Team

文章目录

Introduction

Binder is based on OpenBinder and is an important part of the Android operating system. Binder is built on the Binder driver in the Linux kernel. All the inner-process communication that involves Binder needs to communicate with the Binder driver to transfer data. Due to the important role the Binder driver plays in the entire Android system, it has always been one of the focuses of Android security research.

In the binary world, there are certain processes and time points for object generation and destruction. Some are implemented at the compiler level, while others require programmers to maintain them. Once this process is not designed to be perfect, it can leaves chances for an attacker to disrupt its normal life cycle through illegal operations, thus causing a series security problems. When I was conducting a security audit of the Binder driver code in August last year, I found a killing kernel vulnerability which is as destructive as the "Waterdrop" in "the Three-body Problems". It has great destructive power and can be exploited to implement arbitrary address reading/ writing, arbitrary content writing and information leakage by itself. At the same time, the Binder driver is currently one of the few drivers that can be accessed by processes in the sandbox; hence, the vulnerability could be used for sandbox escaping. It is vital risk since such a killing vulnerability is lurking in such an important kernel driver.

Based on this vulnerability, we gained the root access of all the Pixel series mobile phones. Among them, Pixel 3' operating system represents the highest security defense level of the Google Android kernel. This is the first public ROOT attack since Pixel 3 has been released. I would like to thank @Mingjian Zhou, Xiaodong Wang(@phybio88), Jun Yao(@_2freeman), Dacheng Shao(@DachengSshao) and

A few months later … (II)

project-zero | New issue | Open issues | Search project-zero issues... | Sign in

☆ Starred by 3 users

Owner: jannh@google.com

CC: project-...@google.com

Status: Fixed *(Closed)*

Modified: Mar 6, 2019

Product-Android
Deadline-90
Vendor-Google
Deadline-Grace
Deadline-Exceeded
CCProjectZeroMembers
Severity-High
Finder-jannh
Methodology-source-review
Reported-2018-Nov-23
AndroidID-120023925
CVE-2019-2025
AndroidID-116855682

Issue 1720: Android: binder use-after-free via racy initialization of ->allow_user_free

Reported by jannh@google.com on Fri, Nov 23, 2018, 8:00 PM GMT+1 Project Member

Code

1 of 5
Back to list

The following bug report solely looks at the situation on the upstream master branch; while from a cursory look, at least the wahoo kernel also looks affected, I have only properly tested this on upstream master.

The binder driver permits userspace to free buffers in the kernel-managed shared memory region by using the BC_FREE_BUFFER command. This command implements the following restrictions:

- binder_alloc_prepare_to_free_locked() verifies that the pointer points to a buffer
- binder_alloc_prepare_to_free_locked() verifies that the ->free_in_progress flag is not yet set, and sets it
- binder_thread_write() verifies that the ->allow_user_free flag is set

The first two of these checks happen with alloc->mutex held.

The ->free_in_progress flag can be set in the following places:

- new buffers are allocated with kzalloc() and therefore have the flag set to 0
- binder_alloc_prepare_to_free_locked() sets it to 1 when starting to free a buffer
- binder_alloc_new_buf_locked() sets it to 0 when a buffer is allocated

This means that a buffer coming from binder_alloc_new_buf() always has this flag clear.

The ->allow_user_free flag can be set in the following places:
- new buffers are allocated with kzalloc() and therefore have the flag set to 0
- binder_transaction() sets it to 0 after allocating a buffer with

Comparison with Qihoo360 exploit*

- They use a *binder_buffer* use-after-free
  - Should work from sandbox (no need to control both ends)
  - Requires triggering race and UAF for each r/w (vs 1 race / 2 UAF)
- They use a service that echoes data
  - This service is not reachable from within sandbox
  - Would have to find similar one that's available for a sandbox escape
- They use a Qualcomm-only driver for leaking *struct file* *
  - Doesn't work from sandbox
  - Not generic, but probably easy to replace

* From my understanding of their HITB talk

Comparison with Qihoo360 exploit* (II)

- Played with scheduling and mutexes to improve odds
  - Similar to my colleague’s approach
  - Smarter than mine ;-)
- They also seem to pin CPUs
  - Also not allowed on the sandbox!
- Overall feeling:
  - Ours seems faster and possibly more reliable
  - Theirs has potential to be a sandbox escape too
  - They were first, so they won anyway ;-)

* From my understanding of their HITB talk

Conclusions

- Writing modern exploits for some bugs can be painful
  - Still possible, do not give up
- Fragmentation is annoying
  - Generic bug, lots of corner cases
- Easy to go down a rabbit hole
  - Keep an eye on alternative approaches
  - Always take time to try to find better solutions
- From upstream to Android took months
  - Upstream patch: 6^{th} November 2018
  - Android bulletin: 5^{th} March 2019

1. adb shell (adb)

scrcpy (adb) ⌘1 | adb (adb) ⌘2

```
blueline:/ $ getprop ro.build.id
```

Pixel 3

2:59

Android version

Android version
10

Android security patch level
February 5, 2020

Google Play system update
September 1, 2019

Baseband version
g845-00086-191011-B-5933466

Kernel version
4.9.185-g15c0389f9d0d-ab6076840
#0 Mon Dec 16 20:48:48 UTC 2019

Build number
QQ1A.200205.002

Customise

Thank you!

Blue Frost Security

Eloi Sanfelix
@esanfelix